Troubleshooting: Understanding and Tuning the Shared Pool (Doc ID 62143.1)

In this Document

Purpose
- Introduction
- Ask Questions, Get Help, And Share Your Experiences With This Article

Troubleshooting Steps
- What is the shared pool ?
- Terminology
  - Literal SQL
  - Hard Parse
  - Soft Parse
  - Identical Statements ?
  - Sharable SQL
  - Versions of a statement
  - Library Cache and Shared Pool latches
- Literal SQL versus Shared SQL
  - Literal SQL
- Sharable SQL
- Reducing the load on the Shared Pool
  - Parse Once / Execute Many
  - Eliminating Literal SQL
  - Avoid Invalidations
  - CURSOR_SHARING parameter (8.1.6 onwards)
- SESSION_CACHED_CURSORS parameter
  - CURSOR_SPACE_FOR_TIME parameter
- CLOSE_CACHED_OPEN_CURSORS parameter
  - SHARED_POOL_RESERVED_SIZE parameter
- SHARED_POOL_RESERVED_MIN_ALLOC parameter
- SHARED_POOL_SIZE parameter
- _SQLEXEC_PROGRESSION_COST parameter (8.1.5 onwards)
- Precompiler HOLD_CURSOR and RELEASE_CURSOR Options
- Pinning Cursors in the Shared Pool
- DBMS_SHARED_POOL.KEEP
- Flushing the SHARED POOL
- DBMS_SHARED_POOL.PURGE
- Using V$ Views (V$SQL and V$SQLAREA)
- MTS, Shared Server and XA
- Useful SQL for looking at Shared Pool problems
- Issues in various Oracle Releases
- Bug fixes and Enhancements

Discuss Shared Pool and Library Cache Contention

References

APPLIES TO:

Oracle Database - Personal Edition - Version 7.1.4.0 and later
Oracle Database - Enterprise Edition - Version 7.0.16.0 and later
Oracle Database - Standard Edition - Version 7.0.16.0 and later
Information in this document applies to any platform.

PURPOSE

Introduction

The aim of this article is to introduce the key issues involved in tuning the shared pool in Oracle 7 through 11g. The notes here are particularly important if your system shows any of the following:

- Latch contention for the library cache latch/es or latch:library cache
- Latch contention for the shared pool latch or latch:shared pool
- High CPU parse times
- High numbers of reloads in V$LIBRARYCACHE
- High numbers of versions of cursors
- Lots of parse calls
- Frequent ORA-04031 errors

Ask Questions, Get Help, And Share Your Experiences With This Article

Would you like to explore this topic further with other Oracle Customers, Oracle Employees, and Industry Experts?

Click here to join the discussion where you can ask questions, get help from others, and share your experiences with this specific article.
Discover discussions about other articles and helpful subjects by clicking here to access the main *My Oracle Support Community* page for Database Tuning.

TROUBLESHOOTING STEPS

What is the shared pool ?

Oracle keeps SQL statements, packages, object information and many other items in an area in the SGA known as the shared pool. This sharable area of memory is managed as a sophisticated cache and heap manager rolled into one. It has 3 fundamental problems to overcome:

1. The unit of memory allocation is not a constant - memory allocations from the pool can be anything from a few bytes to many kilobytes
2. Not all memory can be 'freed' when a user finishes with it (as is the case in a traditional heap manager) as the aim of the shared pool is to maximize sharability of information. The information in the memory may be useful to another session - Oracle cannot know in advance if the items will be of any use to anyone else or not.
3. There is no disk area to page out to so this is not like a traditional cache where there is a file backing store. Only "recreatable" information can be discarded from the cache and it has to be re-created when it is next needed.

Given this background one can understand that management of the shared pool is a complex issue. The sections below list the key issues affecting the performance of the shared pool and its associated latches.

Terminology

Literal SQL

A literal SQL statement is considered as one which uses literals in the predicate/s rather than bind variables where the value of the literal is likely to differ between various executions of the statement.
Eg 1:

```
SELECT * FROM emp WHERE ename='CLARK';
```

is used by the application instead of:

```
SELECT * FROM emp WHERE ename=:bind1;
```

TIP: For Additional information on setting up and using bind variables in SQL PLUS, see Using Bind Variables

Eg 2:

```
SELECT sysdate FROM dual;
```

does not use bind variables but would not be considered as a literal SQL statement for this article as it can be shared.

Eg 3:

```
  SELECT version  FROM app_version WHERE version>2.0;
```

If this same statement was used for checking the 'version' throughout the application then the literal value '2.0' is always the same so this statement can be considered sharable.

Hard Parse

If a new SQL statement is issued which does not exist in the shared pool then this has to be parsed fully. Eg: Oracle has to allocate memory for the statement from the shared pool, check the statement syntactically and semantically etc... This is referred to as a hard parse and is very expensive in both terms of CPU used and in the number of latch gets performed.

Soft Parse

If a session issues a SQL statement which is already in the shared pool AND it can use an existing version of that statement then this is known as a 'soft parse'. As far as the application is concerned it has asked to parse the statement.

Identical Statements ?

If two SQL statements mean the same thing but are not identical character for character then from an Oracle viewpoint they are different statements. Consider the following issued by SCOTT in a single session:

```
SELECT ENAME from EMP;

SELECT ename from emp;
```

Although both of these statements are really the same they are not identical as an upper case 'E' is not the same as a lower case 'e'.

Sharable SQL

If two sessions issue identical SQL statements it does NOT mean that the statement is sharable. Consider the following: User SCOTT has a table called EMP and issues:

```
SELECT ENAME from EMP;
```

User FRED has his own table called EMP and also issues:

```
SELECT ENAME from EMP;
```

Although the text of the statements are identical the EMP tables are different objects. Hence these are different versions of the same basic statement. There are many things that determine if two identical SQL strings are truely the same statement (and hence can be shared) including:

- All object names must resolve to the same actual objects
- The optimizer goal of the sessions issuing the statement should be the same
- The types and lengths of any bind variables should be "similar".
  (We don't discuss the details of this here but different types or lengths of bind variables can cause statements to be classed as different versions)
- The NLS (National Language Support) environment which applies to the statement must be the same.

Versions of a statement

As described in 'Sharable SQL' if two statements are textually identical but cannot be shared then these are called 'versions' of the same statement. If Oracle matches to a statement with many versions it has to check each version in turn to see if it is truely identical to the statement currently being parsed. Hence high version counts are best avoided by:

- Standardising the maximum bind lengths specified by the client
- Avoid using identical SQL from lots of different schemas which use private objects. Eg: ***SELECT xx FROM MYTABLE;*** *where each user has their own* ***MYTABLE***
- Setting _SQLEXEC_PROGRESSION_COST to '0' in Oracle 8.1

Library Cache and Shared Pool latches

The shared pool latch is used to protect critical operations when allocating and freeing memory in the shared pool.

The library cache latches (and the library cache pin latch in Oracle 7.1) protect operations within the library cache itself.

All of these latches are potential points of contention. The number of latch gets occurring is influenced directly by the amount activity in the shared pool, especially parse operations. Anything that can minimise the number of latch gets and indeed the amount of activity in the shared pool is helpful to both performance and scalability.

Literal SQL versus Shared SQL

To give a balanced picture this short section describes the benefits of both literal SQL and sharable SQL.

Literal SQL

The Cost Based Optimizer (CBO) works best when it has full statistics and when statements use literals in their predicates. Consider the following:

```
SELECT distinct cust_ref FROM orders WHERE total_cost < 10000.0;
```

versus

```
SELECT distinct cust_ref FROM orders WHERE total_cost < :bindA;
```

For the first statement the CBO could use histogram statistics that have been gathered to decide if it would be fastest to do a full table scan of ORDERS or to use an index scan on TOTAL_COST (assuming there is one). In the second statement CBO has no idea what percentage of rows fall below ":bindA" as it has no value for this bind variable to determine an execution plan . Eg: ":bindA" could be 0.0 or 99999999999999999.9

There could be orders of magnitude difference in the response time between the two execution paths so using the literal statement is preferable if you want CBO to work out the best execution plan for you. This is typical of Decision Support Systems where there may not be any 'standard' statements which are issued repeatedly so the chance of sharing a statement is small. Also the amount of CPU spent on parsing is typically only a small percentage of that used to execute each statement so it is probably more important to give the optimizer as much information as possible than to minimize parse times.

Sharable SQL

If an application makes use of literal (unshared) SQL then this can severely limit scalability and throughput. The cost of parsing a new SQL statement is expensive both in terms of CPU requirements and the number of times the library cache and shared pool latches may need to be acquired and released.

Eg: Even parsing a simple SQL statement may need to acquire a library cache latch 20 or 30 times.

The best approach to take is that all SQL should be sharable unless it is adhoc or infrequently used SQL where it is important to give CBO as much information as possible in order for it to produce a good execution plan.

Reducing the load on the Shared Pool

Parse Once / Execute Many

By far the best approach to use in OLTP type applications is to parse a statement only once and hold the cursor open, executing it as required. This results in only the initial parse for each statement (either soft or hard). Obviously there will be some statements which are rarely executed and so maintaining an open cursor for them is a wasteful overhead.

> **NOTE:** A session only has <Parameter:open_cursors> cursors available and holding cursors open is likely to increase the total number of concurrently open cursors.

In precompilers the HOLD_CURSOR parameter controls whether cursors are held open or not while in OCI developers have direct control over cursors .

Eliminating Literal SQL

If you have an existing application, it is unlikely that you could eliminate all literal SQL but you should be prepared to eliminate some if it is causing problems. By looking at the V$SQLAREA view it is possible to see which literal statements are good candidates for converting to use bind variables. The following queries shows SQL in the SGA where there are a large number of similar statements:

```
SELECT substr(sql_text,1,40) "SQL",
         count(*) ,
         sum(executions) "TotExecs"
    FROM v$sqlarea
   WHERE executions < 5
   GROUP BY substr(sql_text,1,40)
  HAVING count(*) > 30
   ORDER BY 2
  ;
```

The values 40,5 and 30 are example values so this query is looking for different statements whose first 40 characters are the same which have only been executed a few times each and there are at least 30 different occurrences in the shared pool. This query uses the idea it is common for literal statements to begin

```
"SELECT col1,col2,col3 FROM table WHERE ..."
```

with the leading portion of each statement being the same.

Alternatively, from Oracle 10g onwards, the following sql using the FORCE_MATCHING_SIGNATURE column could also be used:

```
SET pages 10000
SET linesize 250
column FORCE_MATCHING_SIGNATURE format 99999999999999999999999
WITH c AS
(SELECT FORCE_MATCHING_SIGNATURE,
COUNT(*) cnt
FROM v$sqlarea
WHERE FORCE_MATCHING_SIGNATURE!=0
GROUP BY FORCE_MATCHING_SIGNATURE
HAVING COUNT(*) > 20
)
,
sq AS
(SELECT sql_text ,
FORCE_MATCHING_SIGNATURE,
row_number() over (partition BY FORCE_MATCHING_SIGNATURE ORDER BY sql_id DESC) p
FROM v$sqlarea s
WHERE FORCE_MATCHING_SIGNATURE IN
(SELECT FORCE_MATCHING_SIGNATURE
FROM c
)
)
SELECT sq.sql_text ,
sq.FORCE_MATCHING_SIGNATURE,
c.cnt "unshared count"
FROM c,
sq
WHERE sq.FORCE_MATCHING_SIGNATURE=c.FORCE_MATCHING_SIGNATURE
AND sq.p =1
ORDER BY c.cnt DESC
```

NOTE(s): The statements above are intended to list example of statements that may be suitable for replacing literals with bind variables. They are not intended to provide definitive lists and, if both are executed, the output is unlikely to be the same.

If there is latch contention for the library cache latches the above statement may cause yet further contention problems.

There is often some degree of resistance to converting literal SQL to use bind variables. Be assured that it has been proven time and time again that performing this conversion for the most frequently occurring statements can eliminate problems with the shared pool and improve scalability greatly.

Refer to the documentation on the tool/s you are using in your application to determine how to use bind variables in statements.

Avoid Invalidations

Some specific orders will change the state of cursors to INVALIDATE. These orders modify directly the context of related objects associated with cursors. That's orders are TRUNCATE, ANALYZE or DBMS_STATS.GATHER_XXX on tables or indexes, grants changes on underlying objects. The associated cursors will stay in the SQLAREA but when it will be reference next time, it should be reloaded and reparsed fully, so the global performance will be impacted.

The following query could help us to better identify the concerned cursors:

```
SELECT SUBSTR(sql_text, 1, 40) "SQL",
invalidations
FROM v$sqlarea
ORDER BY invalidations DESC;
```

To get more details on this, consult Document 115656.1 and Document 123214.1

CURSOR_SHARING parameter (8.1.6 onwards)

<Parameter:cursor_sharing> (introduced in Oracle8.1.6).
It should be used with caution in this release. If this parameter is set to **FORCE** then literals will be replaced by system generated bind variables where possible. For multiple similar statements which differ only in the literals used this allows the cursors to be shared even though the application supplied SQL uses literals. The parameter can be set dynamically at the system or session level thus:

```
ALTER SESSION SET cursor_sharing = FORCE;
```

or

```
ALTER SYSTEM SET cursor_sharing = FORCE;
```

or it can be set in the init.ora file.

NOTE: As the FORCE setting causes system generated bind variables to be used in place of literals, a different execution plan may be chosen by the cost based optimizer (CBO) as it no longer has the literal values available to it when costing the best execution plan.

In Oracle9i, it is possible to set CURSOR_SHARING=SIMILAR. SIMILAR causes statements that may differ in some literals, but are otherwise identical, to share a cursor, unless the literals affect either the meaning of the statement or the degree to which the plan is optimized. This enhancement
improves the usability of the parameter for situations where FORCE would normally cause a different, undesired execution plan. With CURSOR_SHARING=SIMILAR, Oracle determines which literals are "safe" for substitution with bind variables. This will result in some SQL not being shared in an attempt
to provide a more efficient execution plan.

See Document 94036.1 for details of this parameter.

NOTE: Similar will be deprecated in Oracle 12. See:

Document 1169017.1 ANNOUNCEMENT: Deprecating the cursor_sharing = "SIMILAR" setting

SESSION_CACHED_CURSORS parameter

<Parameter:session_cached_cursors> is a numeric parameter which can be set at instance level or at session level using the command:

```
ALTER SESSION SET session_cached_cursors = NNN;
```

The value *NNN* determines how many 'cached' cursors there can be in your session.

Whenever a statement is parsed Oracle first looks at the statements pointed to by your private session cache - if a sharable version of the statement exists it can be used. This provides a shortcut access to frequently parsed statements that uses less CPU and uses far fewer latch gets than a soft or hard parse.

To get placed in the session cache the same statement has to be parsed 3 times within the same cursor - a pointer to the shared cursor is then added to your session cache. If all session cache cursors are in use then the least recently used entry is discarded.

If you do not have this parameter set already then it is advisable to set it to a starting value of about 50. The statistics section of the bstat/estat report includes a value for 'session cursor cache hits' which shows if the cursor cache is giving any benefit. The size of the cursor cache can then be increased or decreased as necessary. SESSION_CACHED_CURSORS are particularly useful with Oracle Forms applications when forms are frequently opened and closed.

CURSOR_SPACE_FOR_TIME parameter

NOTE: CURSOR_SPACE_FOR_TIME has been deprecated in 10.2.0.5 and 11.1.0.7 See:

Document 565424.1 CURSOR_SPACE_FOR_TIME Has Been Deprecated

<Parameter:cursor_space_for_time> controls whether parts of a cursor remain pinned between different executions of a statement. This may be worth setting if all else has failed as it can give some gains where there are sharable statements that are infrequently used, or where there is significant pinning / unpinning of cursors (see <View:v$latch_misses> - if most latch waits are due to "kglpnc: child" and "kglupc: child" this is due to pinning / unpinning of cursors) .

You **must** be sure that the shared pool is large enough for the work load otherwise performance will be badly affected and ORA-4031 eventually signalled.

If you do set this parameter to TRUE be aware that:

- If the SHARED_POOL is too small for the workload then an ORA-4031 is much more likely to be signalled.
- If your application has any cursor leak then the leaked cursors can waste large amounts of memory having an adverse effect on performance after a period of operation.
- There have historically been problems reported with this set to TRUE. The main known issues are:
  - Bug:770924 (Fixed 8061 and 8160) ORA-600 [17302] may occur
  - Bug:897615 (Fixed 8061 and 8160) Garbage Explain Plan over DBLINK
  - Bug:1279398 (Fixed 8162 and 8170) ORA-600 [17182] from ALTER SESSIONSET NLS...

CLOSE_CACHED_OPEN_CURSORS parameter

This parameter has been obsoleted in Oracle8i.

<Parameter:close_cached_open_cursors> controls whether PL/SQL cursors are closed when a transaction COMMITs or not. The default value is FALSE which causes PL/SQL cursors to be kept open across commits which can help reduce the number of hard parses which occur. If this has been set to TRUE then there is an increased chance that the SQL will be flushed from the shared pool when not in use.

SHARED_POOL_RESERVED_SIZE parameter

There are quite a few notes explaining <Parameter:shared_pool_reserved_size> already in circulation. The parameter was introduced in Oracle 7.1.5 and provides a means of reserving a portion of the shared pool for large memory allocations. The reserved area comes out of the shared pool itself.

From a practical point of view one should set SHARED_POOL_RESERVED_SIZE to about 10% of SHARED_POOL_SIZE unless either the shared pool is very large OR SHARED_POOL_RESERVED_MIN_ALLOC has been set lower than the default value:

- If the shared pool is very large then 10% may waste a significant amount of memory when a few **Mb** will suffice.
- If SHARED_POOL_RESERVED_MIN_ALLOC has been lowered then many space requests may be eligible to be satisfied from this portion of the shared pool and so 10% may be too little.

It is easy to monitor the space usage of the reserved area using the <View:v$shared_pool_reserved> which has a column FREE_SPACE.

SHARED_POOL_RESERVED_MIN_ALLOC parameter

In Oracle8i this parameter is hidden.

SHARED_POOL_RESERVED_MIN_ALLOC should generally be left at its default value, although in certain cases values of 4100 or 4200 may help relieve some contention on a heavily loaded shared pool.

SHARED_POOL_SIZE parameter

<Parameter:shared_pool_size> controls the size of the shared pool itself. The size of the shared pool can impact performance. If it is too small then it is likely that sharable information will be flushed from
the pool and then later need to be reloaded (rebuilt). If there is heavy use of literal SQL and the shared pool is too large then over time a lot of small chunks of memory can build up on the internal memory freelists causing the shared pool latch to be held for longer which in-turn can impact performance. In this situation a smaller shared pool may perform better than a larger one. This problem is greatly reduced in 8.0.6 and in 8.1.6 onwards due to the enhancement in Bug:986149 .

NOTE: The shared pool itself should never be made so large that paging or swapping occur as performance can then decrease by many orders of magnitude.

See Document 1012046.6 to calculate the SHARED_POOL_SIZE requirements based on your current workload.

_SQLEXEC_PROGRESSION_COST parameter (8.1.5 onwards)

This is a hidden parameter which was introduced in Oracle 8.1.5. The parameter is included here as the default setting has caused some problems with SQL sharability. Setting this parameter to 0 can avoid these issues which result in multiple versions statements in the shared pool.
Eg: Add the following to the init.ora file

```
        # _SQLEXEC_PROGRESSION_COST is set to ZERO to avoid SQL sharing issues
        # See Document 62143.1 for details
        _sqlexec_progression_cost=0
```

NOTE: A side effect of setting this to '0' is that the V$SESSION_LONGOPS view is not populated by long running queries.

See Document 68955.1 for more details of this parameter.

Precompiler HOLD_CURSOR and RELEASE_CURSOR Options

When using Oracle Precompiler the behavior of the shared pool can be modified by using parameters RELEASE_CURSOR and HOLD_CURSOR when precompiling the program. These parameters will determine the status of a cursor in the library cache and the session cache once the execution of the cursor ends.

For further information on these parameters, please refers to Document 73922.1

Pinning Cursors in the Shared Pool

Another way to alleviate library cache latch is to pin curors in the shared pool. Please refer to following note on how to do this:

Document 130699.1 How to Reduce 'LIBRARY CACHE LATCH' Contention Using a Procedure to KEEP Cursors Executed> 10 times

DBMS_SHARED_POOL.KEEP

This procedure (defined in the DBMSPOOL.SQL script in the RDBMS/ADMIN directory) can be used to KEEP objects in the shared pool. DBMS_SHARED_POOL.KEEP allows one to 'KEEP' packages, procedures, functions, triggers (7.3+) and sequences (7.3.3.1+) and is fully described in Document 61760.1

It is generally desirable to mark frequently used packages such that they are always KEPT in the shared pool. Objects should be KEPT shortly after instance startup since the database does not do it automatically after a shutdown was issued.

NOTE: Prior to Oracle 7.2 DBMS_SHARED_POOL.KEEP does not actually load all of the object to be KEPT into the shared pool. It is advisable to include a dummy procedure in each package to be KEPT. This dummy procedure can then be called after calling DBMS_SHARED_POOL.KEEP to ensure the object is fully loaded. This is not a problem from 7.2 onwards.

Flushing the SHARED POOL

On systems which use a lot of literal SQL the shared pool is likely to fragment over time such that the degree of concurrency which can be achieved diminishes. Flushing the shared pool will often restore performance for a while as it can cause many small chunks of memory to be coalesced. After the flush there is likely to be an interim spike in performance as the act of flushing may remove sharable SQL from the shared pool but does nothing to improve shared pool fragmentation. The command to flush the shared pool is:

```
ALTER SYSTEM FLUSH SHARED_POOL;
```

NOTE(s): Explicitly flushing cursors as above might cause cursors that have been marked as to-be-kept using DBMS_SHARED_POOL.KEEP to be released, together with their associated memory. When the flush is implicit (due to shared pool memory pressure) should not release the "kept" cursor.

Flushing the shared pool will flush any cached sequences potentially leaving gaps in the sequence range. DBMS_SHARED_POOL.KEEP('sequence_name','Q') can be used to KEEP sequences preventing such gaps.

DBMS_SHARED_POOL.PURGE

Single heap can also be flushed instead of the whole shared pool. The following note describes how to purge library cache heaps for 10g and 11g:

Document 751876.1 DBMS_SHARED_POOL.PURGE Is Not Working On 10.2.0.4

Using V$ Views (V$SQL and V$SQLAREA)

Note that some of the V$ views have to take out relevant latches to obtain the data to reply to queries. This is notably so for views against the library cache and SQL area. It is generally advisable to be selective about what SQL is issued against these views. In particular use of V$SQLAREA can place a great load on the library cache latches. Note that V$SQL can often be used in place of V$SQLAREA and can have less impact on the latch gets - this is because V$SQLAREA is a GROUP BY of statements in the shared pool while V$SQL does not GROUP the statements.

MTS, Shared Server and XA

The multi-threaded server (MTS) adds to the load on the shared pool and can contribute to any problems as the User Global Area (UGA) resides in the shared pool. This is also true of XA sessions in Oracle7 as their UGA is located in the shared pool. (*In Oracle8/8i XA sessions do NOT put their UGA in the shared pool*). In Oracle8 the Large Pool can be used for MTS reducing its impact on shared pool activity - However memory allocations in the Large Pool still make use of the "shared pool latch". See Document 62140.1 for a description of the Large Pool.

Using dedicated connections rather than MTS causes the UGA to be allocated out of process private memory rather than the shared pool. Private memory allocations do not use the "shared pool latch" and so a switch from MTS to dedicated connections can help reduce contention in some cases.

In Oracle9i, MTS was renamed to "Shared Server". For the purposes of the shared pool, the behaviour is essentially the same.

Useful SQL for looking at Shared Pool problems

This section shows some example SQL that can be used to help find potential issues in the shared pool. The output of these statements should be spooled to a file.

NOTE: These statements may add to existing latch contention as described in "Using V$ Views (V$SQL and V$SQLAREA)" above.

- **Finding literal SQL**

```
          SELECT substr(sql_text,1,40) "SQL",
                 count(*) ,
                 sum(executions) "TotExecs"
            FROM v$sqlarea
           WHERE executions < 5
           GROUP BY substr(sql_text,1,40)
          HAVING count(*) > 30
           ORDER BY 2
          ;
```

This helps find commonly used literal SQL - See "Eliminating Literal SQL" above.
Another way you might find these is to use the "plan_hash_value" column to group the examples ie:

```
           SELECT substr(sql_text,1,40) "SQL", plan_hash_value,
                 count(*) ,
                 sum(executions) "TotExecs"
            FROM v$sqlarea
           WHERE executions < 5
           GROUP BY plan_hash_value,  substr(sql_text,1,40)
          HAVING count(*) > 30
           ORDER BY 2
           ;
```

- **Finding the Library Cache hit ratio**

```
        SELECT SUM(PINS) "EXECUTIONS",
        SUM(RELOADS) "CACHE MISSES WHILE EXECUTING"
        FROM V$LIBRARYCACHE;
```

  If the ratio of misses to executions is more than 1%, then try to reduce the library cache misses

- **Checking hash chain lengths:**

```
        SELECT hash_value, count(*)
          FROM v$sqlarea
         GROUP BY hash_value
        HAVING count(*) > 5
        ;
```

  This should usually return no rows. If there are any HASH_VALUES with high counts (double figures) then you may be seeing the effects of a bug, or an unusual form of literal SQL statement. It is advisable to drill down and list out all the statements mapping to the same HASH_VALUE. Eg:

```
 SELECT sql_text FROM v$sqlarea WHERE hash_value= ;
```

  and if these look the same get the full statements from **V$SQLTEXT**. It is possible for many literals to map to the same hash value. Eg:*In 7.3 two statements may have the same hash value if a literal value occurs twice in the statement and there are exactly 32 characters between the occurrences.*

- **Checking for high version counts:**

```
        SELECT address, hash_value,
                version_count ,
                users_opening ,
                users_executing,
                substr(sql_text,1,40) "SQL"
          FROM v$sqlarea
         WHERE version_count > 10
        ;
```

  "Versions" of a statement occur where the SQL is character for character identical but the underlying objects or binds etc.. are different as described in "Sharable SQL" above. High version counts can occur in various Oracle8i releases due to problems with progression monitoring. This can be disabled by setting SQLEXEC_PROGRESSION_COST to '0' as described earlier in this note.

- **Finding statement/s which use lots of shared pool memory:**

```
        SELECT substr(sql_text,1,40) "Stmt", count(*),
                sum(sharable_mem)     "Mem",
                sum(users_opening)    "Open",
                sum(executions)       "Exec"
          FROM v$sql
         GROUP BY substr(sql_text,1,40)
        HAVING sum(sharable_mem) > &MEMSIZE
        ;
```

  where *MEMSIZE* is about 10% of the shared pool size in bytes. This should show if there are similar literal statements, or multiple versions of a statements which account for a large portion of the memory in the shared pool.

- **Allocations causing shared pool memory to be 'aged' out**

```
        SELECT *
          FROM x$ksmlru
        WHERE ksmlrnum>0
        ;
```

  NOTE: This select returns no more than 10 rows and then erases the contents of the X$KSMLRU table so be sure to SPOOL the output. The X$KSMLRU table shows which memory allocations have caused the MOST memory chunks to be thrown out of the shared pool since it was last queried. This is sometimes useful to help identify sessions or statements which are continually causing space to be requested. If a system is well behaved and uses well shared SQL, but occasionally slows down this select can help identify the cause. Refer to Document 43600.1 for more information on X$KSMLRU.

Issues in various Oracle Releases

These are some important issues which affect performance of the shared pool in various releases:

- Increasing the CPU processing power of each CPU can help reduce shared pool contention problems in all Oracle releases by decreasing the amount of time each latch is held. A faster CPU is generally better than a second CPU.

- If you have an EVENT parameter set for any reason check with Oracle support that this is not an event that will impact shared pool performance.
- Ensure that there is no shortage of memory available for the Oracle instance so that there is no risk of SGA memory being paged out.

  eg: On AIX shared pool issues may become visible due to incorrect OS configuration - See Document 316533.1 .

Bug fixes and Enhancements

Please refer to the following document for a listing of the main bugs and enhancements affecting the shared pool:

Document 102305.1 Bug Issues Known to Affect the Shared Pool

Discuss Shared Pool and Library Cache Contention

The window below is a live discussion of this article (not a screenshot). We encourage you to join the discussion by clicking the "Reply" link below for the entry you would like to provide feedback on. If you have questions or implementation issues with the information in the article above, please share that below.

10 Respostas Última resposta em 11/05/2017 17:16 por since5.0a

Steve Dixon-Oracle 21/02/2014 15:23

Discussion Thread: Troubleshooting Shared Pool Contenti 62143.1]

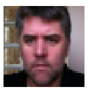
This thread can be used for discussions associated with : Troubleshooting: Tuning the Shared Pool and Tunir [ID 62143.1]

44639 Visualizações **Rótulos:**

Classificação de usuário médio

(1 pontuação)

Sua classificação:



Steve Dixon-Oracle 11/06/2013 08:17 (em resposta a Steve Dixon-Oracle)

1. Re: Discussion Thread: Troubleshooting Shared Pool Contention [Document ID 62143.1]

Does the article meet your needs? Are there any additions that you would find useful?Please post y questions and observations about using Shared Pool or Library Cache Contention or just tips that m users.

Thanks!

REFERENCES

BUG:1366837 - CURSOR NOT SHARED FOR TABLES INVOKING A FUNCTION

NOTE:68955.1 - Init.ora Parameter "_SQLEXEC_PROGRESSION_COST" [Hidden] Reference Note
NOTE:73922.1 - Tuning Precompiler Applications
NOTE:751876.1 - DBMS_SHARED_POOL.PURGE Is Not Working On 10.2.0.4

NOTE:62143.1 - Troubleshooting: Understanding and Tuning the Shared Pool
NOTE:62140.1 - Fundamentals of the Large Pool
NOTE:1012046.6 - How to Calculate Your Shared Pool Size

NOTE:115656.1 - Legacy: Wait Scenarios Regarding 'library cache pin' and 'library cache load lock'

NOTE:1169017.1 - ANNOUNCEMENT: Deprecating the Cursor_Sharing = 'SIMILAR' Setting
NOTE:32871.1 - ALERT: Library Cache Performance Problems in Oracle Releases 7.1.6 to 7.2.3
BUG:1115424 - CURSOR AUTHORIZATION AND DEPENDENCY LISTS GROW CAUSING LATCH CONTENTION

BUG:625806 - CURSOR NOT SHARED FOR VIEWS INVOKING A FUNCTION

BUG:633498 - STATEMENTS IN SHARED POOL DON'T GET REUSED AFTER SELECTING FROM V$OPEN_CURSOR

BUG:1065010 - PERFORMANCE PROBLEM WITH RECURSIVE LINKS
BUG:1484634 - ONE INSTANCE OF OPS HANGS
BUG:1623256 - IDENTICAL SQL REFERENCING SCHEMA.SEQUENCE.NEXTVAL NOT SHARED BY DIFFERENT USERS
BUG:1640583 - ORA-4031 AND CACHE BUFFER CHAIN CONTENTION AFTER UPGRADE TO 8163

NOTE:316533.1 - AIX: Database performance gets slower the longer the database is running

BUG:770924 - ORA-600 [17302][9] DOING A QUIT IN SQLPLUS AFTER CONTEXT QUERY

NOTE:43600.1 - VIEW: X$KSMLRU - LRU flushes from the shared pool - (7.3 - 8.1)
NOTE:61760.1 - Using the Oracle DBMS_SHARED_POOL Package
NOTE:94036.1 - Init.ora Parameter "CURSOR_SHARING" Reference Note

BUG:897615 - EXPLAIN PLAN OVER DBLINK PUTS GARBAGE IN THE PLAN TABLE

NOTE:123214.1 - Truncate - Causes Invalidations in the LIBRARY CACHE

Didn't find what you are looking for?